Safe Compositional Network Sketches: Reasoning with Automated Assistance

Andrei Lapets
Boston University
Boston, MA, USA
lapets@bu.edu

Assaf Kfoury
Boston University
Boston, MA, USA
kfoury@bu.edu

Azer Bestavros
Boston University
Boston, MA, USA
best@bu.edu

Abstract

NetSketch is a tool for the specification of constrained-flow networks (CFNs) and the certification of desirable safety properties imposed thereon, conceived to assist system integrators in modeling and design. It provides compositional analysis capabilities based on a strongly-typed domain-specific language (DSL) for describing and reasoning about CFNs and relevant invariants. Users can model or design individual network components and perform manual or automated whole-system analysis of the properties thereof. Users can also assemble many instances of these components into larger networks, relying on NetSketch's less precise but more tractable compositional analysis capabilities. This ability to trade "precision of analysis" for "feasibility of analysis" according to available resources is among the novel features of NetSketch. In earlier work we illustrated how NetSketch is applied to actual domains [6], and provided a formal definition of its underlying formalism [7].

While the NetSketch DSL provides automatic compositional analysis capabilities for modeling and designing entire networks, users may need to employ a wider variety of tools and techniques when modeling and designing individual network components. These can include common tools for reasoning about systems of constraints of various classes (such as linear constraints, quadratic constraints, and so on), as well as logical systems and ontologies that deal with concepts relevant to the application domain. We integrate the AARTIFACT [14] lightweight automated assistant for formal reasoning (which has also been applied in proving the soundness of the NetSketch formalism [15]) as a tool for modeling and designing individual network components. We present use cases within the context of an example application of the NetSketch DSL that demonstrate how the automated assistant provides NetSketch users with both an interface for reasoning formally about constraints, and a straightforward way to implicitly employ a rich domain-specific ontology of logical propositions. This allows users to verify common properties of constraints and constraint sets, and to reason about constraint relationships using automatically verifiable algebraic manipulations.

1 Introduction and Background

Many large-scale, safety-critical systems can be viewed at least in part as *constrained-flow networks* (CFNs). That is, they can be viewed as interconnections of subsystems, or modules, each of which is a producer, consumer, or regulator of *flows*. These flows are characterized by a set of variables and a set of constraints thereof, reflecting *inherent* or *assumed* properties or rules governing how the modules operate (and what constitutes safe operation). Our notion of flow encompasses streams of physical entities (*e.g.*, vehicles on a road, fluid in a pipe), data objects (*e.g.*, sensor network packets or video frames), or consumable resources (*e.g.*, electric energy or compute cycles).

Current practices in the assembly of such systems involve the integration of various subsystems into a whole by "system integrators" who may not possess expertise or knowledge of the internals of the subsystems on which they rely. NetSketch [6] is a tool for the specification of CFNs and the certification of desirable safety properties imposed on such networks. NetSketch assists system integrators in two types of activities: modeling and design. As a modeling tool, NetSketch enables the abstraction of an existing (flow network) system while retaining sufficient information about it to carry out future

This work is supported in part by NSF awards CNS-0952145, CCF-0820138, CSR-0720604, and EFRI-0735974.

analysis of safety properties. As a design tool, NetSketch enables the exploration of alternative safe designs as well as the identification of minimal requirements for missing subsystems in partial designs. Formal analysis is at the heart of both of the above modeling and design activities. In earlier work we overview NetSketch, highlight its salient features, illustrate how a prototype tool could be used in actual applications [6] and provide a formal definition of its underlying formalism [7].

In related work [7], we present more extensively the background and motivation underlying NetSketch and its underlying formalism, and we refer the reader interested in learning more about this formalism to these materials. Below, we briefly introduce the NetSketch formalism and then discuss the role of an automated assistant, an essential component of any NetSketch implementation.

The NetSketch Formalism. Support for safety analysis in design and/or development tools such as NetSketch is based on sound formalisms that are not specific to (and do not require expertise in) particular domains. NetSketch enables the composition of exact analyses of small subsystems by adopting a constrained-flow network formalism that exposes the tradeoffs between exactness of analysis and scalability of analysis. This is done using a strongly-typed domain-specific language (DSL) for describing and reasoning about CFNs at various levels of "sketchiness" (precision) along with invariants that need to be enforced thereupon. This DSL has a formal definition, and its type system has been proven sound relative to appropriately-defined notions of validity [7].

There are a few essential concepts that constitute our formalism for compositional analysis of problems involving constrained-flow networks. The formalism provides a language for defining *networks* – graphs consisting of nodes and edges. Networks are constructed out of many instances of defined *modules* (small network components). These are also graphs but are typically of a size that is sufficiently small for a complete or exhaustive analysis. The formalism provides means for defining such modules and assembling them into networks. If an analysis of the network is desired, it may be possible to take advantage of this modularity by analyzing the components individually in a more precise manner (using an automated assistant, as discussed further below), and then composing the results to analyze the entire network (using the compositional analysis capabilities of the formalism). Analyses are represented using a language of *constraints*. If the engineer considers flows across a network of modules, the relevant *parameters* describing this flow (e.g., the number of open lanes on a highway, or the throughput of a network link) can be mathematically constrained for each instance of a module. These constraints can model both limitations of modules and problems the engineer must solve.

Thus, the formalism consists of two intertwined languages: one for describing networks composed of modules, and another for describing constraints governing flows across the network components. In order to ensure that our system works correctly "under the hood", a precise *semantics* for these languages is defined, along with a rigorous notion of what it means for an analysis of a network to be "correct".

Module Design and Analysis with an Automated Assistant. The NetSketch DSL and formalism provide a means by which to perform compositional analysis of entire networks of modules (the NetSketch "Sketch Mode" in Fig. 1), and in Section 2 we present the inference rules that enable this capability. However, the formalism's compositional analysis capabilities do not address the issue of individual module construction (the NetSketch "Base Mode" in Fig. 1). The latter involves manipulation of common mathematical concepts such as dimensions, sets, graphs, constraints, and (in)equations. Furthermore, the constraints commonly found in a particular domain might be of certain classes (such as linear constraints, quadratic constraints, and so on), and thus could effectively be analyzed and solved by particular tools and techniques from a broader collection of tools for solving systems of constraints of various classes. Consequently, the tool must leverage a varied collection of existing systems and formalisms. There are many reference textbooks and survey papers to draw on given the task of assembling and using such tools [17, 12, 23, 13]).

An *automated assistant* can help a user manage these formal and mathematical concepts when modeling or assembling individual modules. The automated assistant provides an accessible and lightweight interface that allows the user to (1) represent and manipulate common concepts in the domain of application, (2) access and leverage an extensive ontology of definitions and facts about these concepts, (3) utilize a variety of tools and algorithms that operate on these concepts.

In this work, we integrate the NetSketch DSL with AARTIFACT [1], a lightweight automated assistant for formal reasoning [14] (which has also been applied in proving the soundness of the NetSketch formalism [15]). The AARTIFACT system provides a familiar concrete syntax for common mathematical concepts that overlaps with English, MediaWiki markup, and LaTeX. The system's flexible parser allows the user to employ a selection of LaTeX constructs for mathematical notation, to use predicates represented as natural language phrases, and to introduce her own constants and infix operators. It is the automated assistant's responsibility to recognize whether a particular manipulation is valid based on the system's ability to utilize the algorithms, underlying tools, and ontologies integrated within it.

An automated assistant for formal reasoning can provide several useful capabilities (see Fig. 1). It can provide an interface for the variety of tools a domain expert may want to employ while modeling or assembling CFNs. Specifically, it can leverage domain-specific ontologies of relevant propositions and concepts in helping a domain expert manipulate and reason about constraint collections. This includes (1) checking the sorts (i.e. dimensions) of subexpressions within complex constraints, (2) identifying the classes to which a constraint belongs (and invoking appropriate solvers), (3) reasoning algebraically about relationships between constraints (with support for a symbolic manipulation and evaluation).

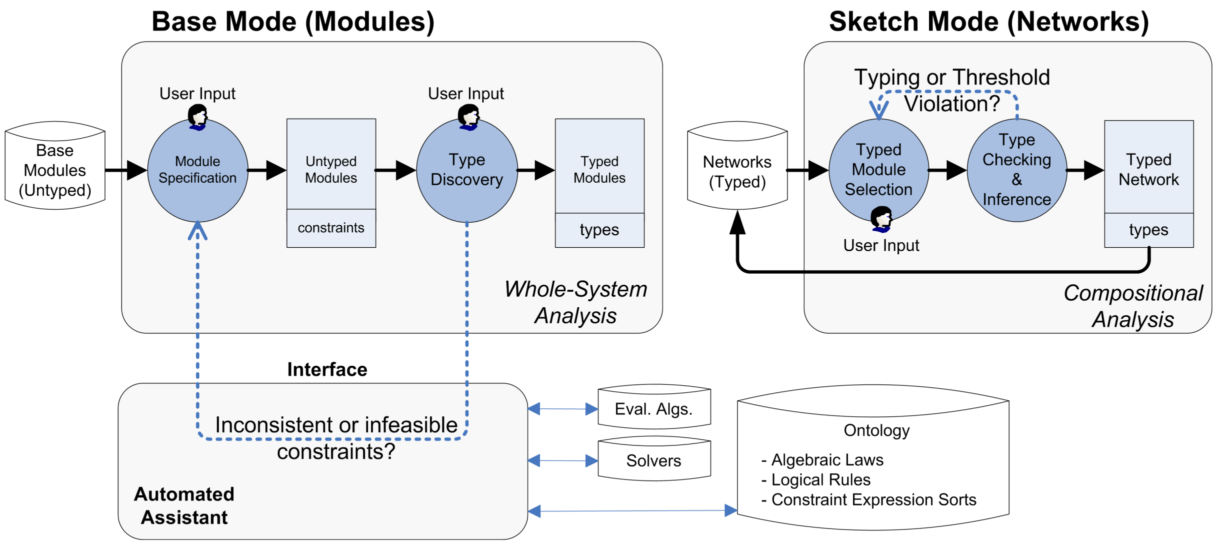

Figure 1: The two-tiered NetSketch modelling and assessment process, with automated assistant support.

2 Overview of the NetSketch Formalism

We describe the generic formal framework underlying the NetSketch DSL; a more detailed presentation can be found in previous work [5].

[1]Source code (for a Haskell implementation) and a demonstration version of the general-purpose automated assistant, integrated with the MediaWiki online content management system, is available at `http://www.aartifact.org`.

$\mathscr{M},\mathscr{N},\mathscr{P} \in \textsc{RawCFNs}$	$::=$	$\mathscr{A}$	module name
	$\mid$	X	hole name
	$\mid$	$\textbf{conn}(\theta,\mathscr{M},\mathscr{N})$	$\theta \subseteq_{1\text{-}1} \mathsf{Out}(\mathscr{M}) \times \mathsf{In}(\mathscr{N})$
	$\mid$	$\textbf{loop}(\theta,\mathscr{M})$	$\theta \subseteq_{1\text{-}1} \mathsf{Out}(\mathscr{M}) \times \mathsf{In}(\mathscr{M})$
	$\mid$	$\textbf{let}\ X \in \{\mathscr{M}_1,\ldots,\mathscr{M}_n\}\ \textbf{in}\ \mathscr{N}$	X occurs once in $\mathscr{N}$

Figure 2: Syntax of raw untyped CFNs.

Constraints and Types. We denote by $\mathbb{N}$ the set of natural numbers. $\mathscr{X} = \{x_0, x_1, x_2, \ldots\}$ is a countably infinite set of *parameters*. The set of *expressions* and *constraints* over $\mathbb{N}$ and $\mathscr{X}$ can be defined using an extended BNF notation, in a style borrowed from the foundations of programming languages:

$$\begin{aligned} e \in \textsc{Exp} &::= n \mid x \mid e_1 * e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots \\ c \in \textsc{Const} &::= e_1 = e_2 \mid e_1 < e_2 \mid e_1 \leqslant e_2 \mid \ldots \end{aligned}$$

While we restrict $\textsc{Const}$ to equalities and inequalities, future extensions will introduce more complex constraints.[2] Of special interest are *linear constraints*, obtained by restricting the definitions for $\textsc{Const}$:

$$\begin{aligned} e \in \textsc{LinExp} &::= n \mid x \mid n * x \mid e_1 + e_2 \\ c \in \textsc{LinConst} &::= e_1 = e_2 \mid e_1 < e_2 \mid e_1 \leqslant e_2 \end{aligned}$$

Constraints in $\textsc{Const}$ characterize the internal workings of a system, be it at the level of a small module or a large network. Such constraints reflect what we call the *internal* safety-critical properties of the system. Constraints in $\textsc{LinConst}$ are to be *inferred* and/or *checked* against the internal constraints.

Depending on the application, the parameters in $\mathscr{X}$ can be associated with particular *sorts* expressed in relevant units of measurement. For example, in modeling vehicular traffic networks, $\mathscr{X}$ has parameters of sorts such as traffic *velocity* (km/hr) and traffic *density* (T/km). An automated assistant can then infer the sorts of expressions, as described in Section 3.

Types (*i.e.*, *boundary constraints*, be it for modules, subsystems, or whole systems) are drawn from $\textsc{LinConst}$ (or a restricted and more tractable subset thereof). Types can be as simple as intervals, *e.g.*, $n_1 \leqslant x \leqslant n_2$ for some natural numbers n_1 and n_2. For brevity, we do not distinguish between types and other constraints in $\textsc{LinConst}$. Types should be simple enough so that their manipulation is not prohibitively costly, but expressive enough that their satisfaction ensures some desirable safety invariants.

Modules and Holes. We specify an *untyped module* $\mathscr{A}$ by a four-tuple $(\mathscr{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})$ where $\mathscr{A}$ is the *module name* and In, Out, and Con are the finite sets of *input parameters*, *output parameters* and *internal constraints* of $\mathscr{A}$, respectively. We require that $\mathsf{In} \cap \mathsf{Out} = \varnothing$ and $\mathsf{In} \cup \mathsf{Out} \subseteq \mathsf{parameters}(\mathsf{Con})$, where $\mathsf{parameters}(\mathsf{Con})$ is the set of parameters occurring in Con, possibly larger than $\mathsf{In} \cup \mathsf{Out}$. An *untyped network hole* is a triple: $(X, \mathsf{In}, \mathsf{Out})$ where X is the *hole name*, In is a finite set of *input parameters*, and Out is a finite set of *output parameters*. There are no internal constraints associated with a hole.

We are careful in adding the name of the module, $\mathscr{A}$, to its specification; in our formal setup we want to be able to refer to the module by its name without the overhead of the rest of its specification. By a slight abuse of notation, we may write informally $\mathscr{A} = (\mathscr{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})$ and $X = (X, \mathsf{In}, \mathsf{Out})$.

Untyped CFNs. An untyped CFN is written as $(\mathscr{M}, I, O, \mathscr{C})$, where I and O are the sets of input and output parameters, and $\mathscr{C}$ is a finite set of finite constraint sets. $\mathscr{M}$ is *not* a name but an expression

[2] These can include *conditional*, *negated*, or *time-dependent* constraints, and others, possibly over domains other than $\mathbb{N}$.

built up from: (1) module names, (2) hole names, and (3) the constructors **conn**, **loop** and **let-in**.[3] We may refer to such a CFN by just writing the expression $\mathscr{M}$ or, more fully, by writing it with its three omitted parts $(\mathscr{M}, I, O, \mathscr{C})$ – the context dictates which of the two to use. For such an untyped CFN $\mathscr{M}$, we define $\mathsf{In}(\mathscr{M})$ as I (the omitted input parameters) and $\mathsf{Out}(\mathscr{M})$ as O (the omitted output parameters). More precisely, the syntax of *raw untyped CFNs* is given in extended BNF in Figure 2. We write $\theta \subseteq_{1\text{-}1} \mathsf{Out}(\mathscr{M}) \times \mathsf{In}(\mathscr{N})$ to denote a partial one-one map from $\mathsf{Out}(\mathscr{M})$ to $\mathsf{In}(\mathscr{N})$. (If the set of parameters is sorted with more than one sort then θ must respect sorts.

In an expression "**let** $X \in \{\mathscr{M}_1, \ldots, \mathscr{M}_n\}$ **in** $\mathscr{N}$ ", we call "$X \in \{\mathscr{M}_1, \ldots, \mathscr{M}_n\}$" a *binding* for the hole X and "$\mathscr{N}$" the *scope* of this binding. Informally, the idea is that all of the CFNs in $\{\mathscr{M}_1, \ldots, \mathscr{M}_n\}$ can be interchangeably placed in the hole X, depending on changing conditions of operation in the CFN as a whole. If a hole X occurs in a CFN $\mathscr{M}$ outside the scope of any **let**-binding, we say X is *free* in $\mathscr{M}$. If there are no free occurrences of holes in $\mathscr{M}$, we say that $\mathscr{M}$ is *closed*.

The expressions defined by the preceding BNF syntax are said to specify "raw" CFNs because they ignore how input parameters I, output parameters O, and internal constraints $\mathscr{C}$ of a CFN $\mathscr{M}$ are assembled from those of its constituent parts. There are formal rules for this purpose, shown in Figure 3. Justification and further elaboration of these rules are in our report [5]. In rules CONNECT and LET, the parameters of the constituent components – $\mathscr{M}$ and $\mathscr{N}$ in CONNECT, $\mathscr{M}_1, \ldots, \mathscr{M}_n$ and $\mathscr{N}$ in LET – are pairwise disjoint sets; this is achieved by parameter renaming (already indicated by the side-condition of rule MODULE) so that each separate component in a CFN has its own private set of parameters. Note there is no need to freshly rename the parameters of a hole X, in rule HOLE, because we assume X occurs exactly once in a CFN (in this presentation), whereas we impose no such restriction on the number of occurrences of a module $\mathscr{A}$.

Rule CONNECT takes two untyped CFNs, $\mathscr{M}$ and $\mathscr{N}$, and returns an untyped CFN $\mathbf{conn}(\theta, \mathscr{M}, \mathscr{N})$ where some of the output parameters in $\mathscr{M}$ are unified with some of the input parameters in $\mathscr{N}$, according to what θ prescribes. Rule LOOP takes one untyped CFN, $\mathscr{M}$, and returns an untyped CFN $\mathbf{loop}(\theta, \mathscr{M})$ where some output parameters in $\mathscr{M}$ are identified with some input parameters in $\mathscr{M}$ according to θ.

Rule LET allows any of $n \geqslant 1$ untyped CFNs $\{\mathscr{M}_1, \ldots, \mathscr{M}_n\}$ to be placed in a hole X of an untyped CFN $\mathscr{N}$. Each $\mathscr{M}_k$, for $1 \leqslant k \leqslant n$, which is placed in X gives rise to a different set of internal constraints for $\mathscr{N}$. (This explains why the specification of an untyped CFN involves a *finite collection* $\mathscr{C}$ of finite constraint sets rather than a *single* finite constraint set.) In the side-condition of rule LET, the maps φ and ψ are bijections. If parameters are multi-sorted, then φ and ψ must respect sorts, *i.e.*, if $\varphi(x) = y$ then both x and y must be of the same sort, *e.g.*, both velocity parameters, or both density parameters, etc., and similarly for ψ.[4]

Pre- and Post-Conditions. A *typing* or a *typed specification* for an untyped CFN $\mathscr{M}$ is an expression of the form $(\mathscr{M} : C^*)$ or, if fully spelled out, $\big((\mathscr{M}, I, O, \mathscr{C}) : C^*\big)$ where C^* is a finite set of *linear* constraints over $\mathsf{In}(\mathscr{M}) \cup \mathsf{Out}(\mathscr{M}) = I \cup O$. As it stands, a typing $(\mathscr{M} : C^*)$ may or may not be valid. The validity of typed specifications presumes a formal definition of the semantics of CFNs, which we discuss after the rules for building typed CFNs.

[3]We may add other constructors according to need. There are also alternative constructors. For example, instead of **conn**, we may introduce **par** (for *parallel* composition of two CFN's). Conversely, **par** can be expressed using **conn** (using $\theta = \varnothing$).

[4]**Caution:** What we define here is a DSL (*d*omain *s*pecific *l*anguage) for design and analysis of large-scale CFNs. The presence of the **let-in** constructor is reminiscent of a similarly-named feature in some programming languages. However, beyond the name, the two features are different. In our DSL there is no notion of **let-in** *reduction* whereby every **let-in** is eliminated by a process of repeated substitutions – although this is possible it also leads to an exponential explosion in the size of what we envision as already large CFN's in practice. Our DSL is a specification language, not a programming language to be compiled into executable code.

$$\textsc{Hole}\quad \frac{(X,\mathsf{In},\mathsf{Out})\ \in\ \Gamma}{\Gamma\vdash(X,\mathsf{In},\mathsf{Out},\{\ \})}$$

$$\textsc{Module}\quad \frac{(\mathscr{A},\mathsf{In},\mathsf{Out},\mathsf{Con})\ \text{module}}{\Gamma\vdash(\mathscr{B},I,O,\{C\})}\qquad (\mathscr{B},I,O,C)=\grave{}(\mathscr{A},\mathsf{In},\mathsf{Out},\mathsf{Con})\ \text{where}\ \grave{}(\ \ )\ \text{denotes a parameter-renaming operator}$$

$$\textsc{Connect}\quad \frac{\Gamma\vdash(\mathscr{M},I_1,O_1,\mathscr{C}_1)\qquad \Gamma\vdash(\mathscr{N},I_2,O_2,\mathscr{C}_2)}{\Gamma\vdash(\mathbf{conn}(\theta,\mathscr{M},\mathscr{N}),I,O,\mathscr{C})}\qquad \begin{array}{l}\theta\subseteq_{\text{1-1}}O_1\times I_2, I=I_1\cup(I_2-\mathsf{range}(\theta)), O=(O_1-\mathsf{domain}(\theta))\cup O_2\\ \mathscr{C}=\{C_1\cup C_2\cup\{\,p=q\,|\,(p,q)\in\theta\,\}\,|\,C_1\in\mathscr{C}_1, C_2\in\mathscr{C}_2\}\end{array}$$

$$\textsc{Loop}\quad \frac{\Gamma\vdash(\mathscr{M},I_1,O_1,\mathscr{C}_1)}{\Gamma\vdash(\mathbf{loop}(\theta,\mathscr{M}),I,O,\mathscr{C})}\qquad \begin{array}{l}\theta\subseteq_{\text{1-1}}O_1\times I_1,\ I=I_1-\mathsf{range}(\theta),\ O=O_1-\mathsf{domain}(\theta)\\ \mathscr{C}=\ \{C_1\cup\{\,p=q\,|\,(p,q)\in\theta\,\}\,|\,C_1\in\mathscr{C}_1\}\end{array}$$

$$\textsc{Let}\quad \frac{\Gamma\vdash(\mathscr{M}_k,I_k,O_k,\mathscr{C}_k)\quad \text{for } 1\leqslant k\leqslant n\qquad \Gamma\cup\{(X,\mathsf{In},\mathsf{Out})\}\vdash(\mathscr{N},I,O,\mathscr{C})}{\Gamma\vdash(\ \mathbf{let}\,X\in\{\mathscr{M}_1,\ldots,\mathscr{M}_n\}\ \mathbf{in}\ \mathscr{N}\ ,I,O,\mathscr{C}'\ )}$$

$$\mathscr{C}'=\Big\{C\cup\hat{C}\cup\{\,p=\varphi(p)\,|\,p\in I_k\,\}\cup\{\,p=\psi(p)\,|\,p\in O_k\,\}\,\Big|\,1\leqslant k\leqslant n, C\in\mathscr{C}, \hat{C}\in\mathscr{C}_k, \varphi: I_k\to\mathsf{In}, \psi: O_k\to\mathsf{Out}\Big\}$$

Figure 3: Specifying the input parameters I, output parameters O, and internal constraints $\mathscr{C}$.

Let C be a finite set of constraints and $\mathscr{Y}=\mathsf{parameters}(C)$. A valuation V of $\mathscr{Y}$ is a total map from $\mathscr{Y}$ to $\mathbb{N}$. *Satisfaction* of C by V is defined in the usual way and written $V\models C$. Let C' be another finite set of constraints over $\mathscr{Y}$. We say C *implies* C', in symbols $C\Rightarrow C'$, if every valuation satisfying C also satisfies C'. The *closure* of C is the set of all constraints implied by C, *i.e.*, $\mathsf{closure}(C)=\{c\in\textsc{Const}\,|\,C\Rightarrow\{c\}\}$. Let C be a constraint set and $\mathscr{X}$ a set of parameters. We define two restrictions of C relative to $\mathscr{X}$:

$$C\restriction\mathscr{X}=\{\,c\in C\,|\,\mathsf{parameters}(c)\subseteq\mathscr{X}\,\}\quad \textit{and}\quad C\downharpoonright\mathscr{X}=\{\,c\in C\,|\,\mathsf{parameters}(c)\cap\mathscr{X}\neq\varnothing\,\}.$$

That is, $(C\restriction\mathscr{X})$ is the set of constraints in C where only parameters from $\mathscr{X}$ occur, and $(C\downharpoonright\mathscr{X})$ is the set of constraints in C with at least one occurrence of a parameter from $\mathscr{X}$. Let $\mathscr{M}=(\mathscr{M},I,O,\mathscr{C})$ be an untyped CFN and $(\mathscr{M}:C^*)$ a typing for $\mathscr{M}$. We partition $\mathsf{closure}(C^*)$ into two subsets as follows:

$$\mathsf{pre}(\mathscr{M}:C^*)=\mathsf{closure}(C^*)\restriction I\quad \textit{and}\quad \mathsf{post}(\mathscr{M}:C^*)=\mathsf{closure}(C^*)-\mathsf{pre}(\mathscr{M}:C^*)=\mathsf{closure}(C^*)\downharpoonright O$$

While the parameters of $\mathsf{pre}(\mathscr{M}:C^*)$ are all in I, the parameters of $\mathsf{post}(\mathscr{M}:C^*)$ are not necessarily all in O, as some constraints in C^* may contain both input and output parameters. The pre-condition of $(\mathscr{M}:C^*)$ involves only input parameters; the post-condition of $(\mathscr{M}:C^*)$ may involve both input and output parameters.[5]

Typed CFNs. We define typed specifications by the same inference rules we already used to derive untyped specifications, but now augmented with type information. To save space here, we show two of these type-augmented rules, CONNECT and LET, together with an additional one WEAKEN in Figure 4. As the exhibited CONNECT and LET elaborate the rules by the same names in Figure 3, we omit some of the side conditions and mention only the parts that are necessary for inserting the types. Further explanation about these rules and the omitted rules are in our report [5].

In Figure 4, we place the crucial side-condition for each rule in a framed box; this condition expresses a relationship that must be satisfied by the "derived types" (linear constraints) in the premises of the rule.

For later reference, we call this side-condition (Ct) in CONNECT, (Lt) in LET, and (Wn) in WEAKEN.

[5]Both $\mathsf{pre}(C^*)$ and $\mathsf{post}(C^*)$ are infinite sets. In an implementation, we need to efficiently compute "canonical finite bases" for $\mathsf{pre}(C^*)$ and $\mathsf{post}(C^*)$ in order to then efficiently decide questions such as: is a certain type c in $\mathsf{pre}(C^*)$, *i.e.*, implied by the latter's finite basis? Does $\mathsf{post}(C^*)$ imply $\mathsf{pre}(C^*)$, *i.e.*, does the finite basis of the first imply that of the second? And others.

$$\textsc{Connect}\quad \frac{\Gamma \vdash (\mathscr{M}, I_1, O_1, \mathscr{C}_1) : C_1^* \qquad \Gamma \vdash (\mathscr{N}, I_2, O_2, \mathscr{C}_2) : C_2^*}{\Gamma \vdash (\textbf{conn}(\theta, \mathscr{M}, \mathscr{N}), I, O, \mathscr{C}) : C^*}$$

$\theta \subseteq_{1\text{-}1} O_1 \times I_2,\ I = I_1 \cup (I_2 - \mathsf{range}(\theta)),\ O = (O_1 - \mathsf{domain}(\theta)) \cup O_2,\ C^* = (C_1^* \cup C_2^*) \upharpoonright (I \cup O),$

(Ct) $\boxed{\mathsf{post}(\mathscr{M} : C_1^*) \Rightarrow \{x = y \mid (x,y) \in \theta\} \cup (\mathsf{pre}(\mathscr{N} : C_2^*) \downharpoonleft \mathsf{range}(\theta))}$

$$\textsc{Let}\quad \frac{\Gamma \vdash (\mathscr{M}_k, I_k, O_k, \mathscr{C}_k) : C_k^* \quad \text{for } 1 \leqslant k \leqslant n \qquad \Gamma \cup \{(X, \mathsf{In}, \mathsf{Out}) : \mathsf{Con}^*\} \vdash (\mathscr{N}, I, O, \mathscr{C}) : C^*}{\Gamma \vdash \big(\, \textbf{let}\, X \in \{\mathscr{M}_1, \ldots, \mathscr{M}_n\}\ \textbf{in}\ \mathscr{N}, I, O, \mathscr{C}'\,\big) : C^*}$$

for all $1 \leqslant k \leqslant n$ and pairs of bijections $(\varphi, \psi) : (I_k, O_k) \to (\mathsf{In}, \mathsf{Out})$:

(Lt) $\boxed{C_k^* \Leftrightarrow \big(\mathsf{Con}^* \cup \{x = \varphi(x) \mid x \in I_k\} \cup \{x = \psi(x) \mid x \in O_k\}\big)}$

$$\textsc{Weaken}\quad \frac{\Gamma \vdash (\mathscr{M}, I, O, \mathscr{C}) : C_1^*}{\Gamma \vdash (\mathscr{M}, I, O, \mathscr{C}) : C^*} \quad (\text{Wn}) \quad \boxed{\mathsf{pre}(\mathscr{M} : C_1^*) \Leftarrow \mathsf{pre}(\mathscr{M} : C^*) \ \text{ and } \ \mathsf{post}(\mathscr{M} : C_1^*) \Rightarrow \mathsf{post}(\mathscr{M} : C^*)}$$

Figure 4: Three of the typing rules for CFNs.

There are different versions of rule LET depending on the side condition (Lt) – the weaker the side condition, the more powerful the rule, *i.e.*, the more CFNs for which it can derive a type. The simplest way of formulating LET, as shown in Figure 4, makes (Lt) most restrictive. However, in the presence of WEAKEN, (Lt) is far less restrictive than it appears; it allows to adjust the derived types and constraints of the CFNs in $\{\mathscr{M}_1, \ldots, \mathscr{M}_n\}$ in order to satisfy (Lt), if possible by weakening them. Rule WEAKEN requires the *pre-condition be strengthened* and/or the *post-condition be weakened* in order that the typing of $\mathscr{M}$ be weakened.

Validity and Soundness. There are different ways of defining the *satisfaction* of a typed specification $(\mathscr{M} : C^*)$ – or, fully spelled out, $\big((\mathscr{M}, I, O, \mathscr{C}) : C^*\big)$ – by a valuation V of I. We choose a particular way that is suitable for purposes of illustration. Another, called *weak satisfaction*, is in our report [5], and there are others still, depending on the applications.
We say valuation V ***satisfies*** $(\mathscr{M} : C^*)$ and write $V \models (\mathscr{M} : C^*)$ to mean:

- if $V \models \mathsf{pre}(C^*)$ then, for every valuation $V' \supseteq V$ of $I \cup O \cup \mathsf{parameters}(\mathscr{C})$ and for every $C \in \mathscr{C}$, if $V' \models C$ then $V' \models \mathsf{post}(C^*)$.

Informally, V satisfies $(\mathscr{M} : C^*)$ in case, if V satisfies $\mathsf{pre}(C^*)$ and V' is an extension of V satisfying the internal constraints of $\mathscr{M}$, then V' satisfies $\mathsf{post}(C^*)$. We say $(\mathscr{M} : C^*)$ is ***valid*** if:

- for every valuation V of I, it holds that $V \models (\mathscr{M} : C^*)$, in which case we also write $\models (\mathscr{M} : C^*)$.

Informally, $(\mathscr{M} : C^*)$ is valid in case, for every input flow satisfying $\mathsf{pre}(C^*)$ and *for every way* of channelling the flow through $\mathscr{M}$, consistent with its internal constraints, $\mathsf{post}(C^*)$ is satisfied. The following ***soundness*** result shows that derivability of a typed specification implies its validity:[6]

- if $\Gamma \vdash \mathscr{M} : C^*$ can be derived by the typing rules (three of them shown in Fig. 4), then $\models (\mathscr{M} : C^*)$.

The proof of this result [5] holds for the particular notion of validity defined above .

[6]This is similar to a type system for a strongly-typed programming language. A *sound* type system is one that is not too lenient: it rejects all invalid programs that violate desirable invariant properties (possibly in addition to some valid programs). A *complete* type system accepts all valid programs, possibly in addition to some invalid ones. For non-trivial (*i.e.*, Turing-complete) programming languages, type systems are sound but not complete.

3 A NetSketch Use Case and the Automated Assistant

We consider an example of a CFN scenario suitable for analysis using the NetSketch DSL: a *vehicular traffic* problem, inspired by problems in the study of traffic flow [20]. We focus on how an automated assistant can be useful for an engineer who is modeling *modules* in a domain such as this. A more extensive examination of this example, including how the engineer in this scenario would assemble *networks* out of many typed module instances, is examined more in a pertinent report [5]. The examples in this discussions are simple and only illustrate the use of an automated assistant; more extensive and sophisticated examples can be found in the full technical report describing this work [16].

Format of Presentation. Formal definitions and arguments parsed by the AARTIFACT system are presented within a framed box (example provided below).

Assert for any $x, y, z \in \mathbb{R}$, if $x > y$ and $y > z$ then $x > z$.

Traffic Flow Example. An engineer working for a metropolitan traffic authority has the following problem. Her city lies on a river bank across from the suburbs, and every morning hundreds of thousands of motorists drive across a few bridges to work in the city center. Each bridge has a fixed number of lanes, but they are all reversible (an operator can decide how many lanes are available to inbound and outbound traffic at different times of the day). The engineer must decide how many inbound lanes should be open in the morning with the goal of ensuring that no backups occur within the city center, with the secondary goal of maximizing the amount of traffic that can enter the city. The city street grid is a network of a large number of only a few distinct kinds of traffic junctions, such as the *merge* junction described below. The junction and the problem the engineer must solve locally within it is modelled as a module.

Derivation of Sorts for Constraint Subexpressions. The capabilities of an automated assistant are a strict superset of the capabilities of a type inference algorithm, so it is natural to task the automated assistant with deriving and verifying the sorts of any expressions within the constraints introduced by the user. In order to model the notion of "no backups", it is necessary to utilize two sorts of parameters: *density* measured in tons per kilometer (T/km), and *velocity* measured in kilometers per hour (km/hr).

We consider an example of a module. A *merge* junction has two incoming links (call them 1 and 2) and one outgoing link (call it 3). This traffic junction can be modelled using a module (call it $\mathscr{A}_{\mathbf{M}}$) consisting of a single node (call it $\mathbf{M}$) for which the formal definition is:

$$\begin{aligned} N &= \{\mathbf{M}\}, \mathsf{In} = \{1,2\}, Q = \varnothing, \mathsf{Out} = \{3\} \\ \mathsf{Par} &= \{1 \mapsto (v_1, d_1), 2 \mapsto (v_2, d_2), 3 \mapsto (v_3, d_3)\} \\ \mathsf{Con} &= \{\text{regulating traffic through } \mathbf{M}\} \cup \{\text{lower/upper bounds on } v_i, d_i\} \end{aligned}$$

This definition specifies the structure of the junction, with the constraint set Con only containing predefined constraints that are required for all *merge* junctions:

$$(1)\ d_1 + d_2 = d_3; \quad (2)\ d_1 * v_1 + d_2 * v_2 \leq d_3 * v_3$$

Constraint (1) enforces *conservation of density* when $\mathbf{M}$ is neither a "sink" nor a "source", whereas constraint (2) encodes the *non-decreasing flow* invariant, namely that traffic along the exit link may accelerate or become more dense.

When the two constraints are introduced, the automated assistant automatically derives sorts for each subexpression, as well as the correctness (and thus meaningfulness) of the overall constraint. For example, because the dimension associated with both d_1 and d_2 is T/km, the dimension associated with $d_1 + d_2$ is also T/km. Because T/km is the dimension associated with the expressions on both sides of the constraint, (1) is validated as a well-sorted constraint. Furthermore, in constraint (2), because d_2 is T/km and v_2 is km/hr, $v_2 * d_2$ is T/hr (called *mass flow*).

Symbolic Verification of Relationships between Constraints. Suppose that the user introduces two more constraints that represent an upper bound k on the increase in traffic density on the outgoing link:

$$(3)\ d_3 \leq d_1 + k; \quad (4)\ d_3 \leq d_2 + k.$$

Together with constraints (1) and (2), these constraints have some noteworthy consequences for the module **M** that can be derived using a few algebraic manipulations:

Assert $d_3 \leq d_1 + k$, $d_1 + d_2 \leq d_1 + k$, and $d_2 \leq k$.
Assert $d_3 \leq d_2 + k$, $d_1 + d_2 \leq d_2 + k$, and $d_1 \leq k$.

Thus, these constraints in fact put an upper bound on the traffic density over incoming links. Now, suppose the user introduces two more constraints,

$$(5)\ d_3 \geq d_1 + d_2 \quad (6)\ v_3 \geq v_1 + v_2,$$

and wishes to know whether these guarantee that kinetic energy (km $\cdot$ T/hr^2) is preserved:

$$(7)\ v_3^2 * d_3 \geq v_1^2 * d_1 + v_1^2 * d_2.$$

This, too, can be verified using a sequence of algebraic manipulations.

Assert $v_3^2 * d_3 \;\geq\; (v_1 + v_2)^2 * d_3$,
$(v_1 + v_2)^2 * d_3 \;\geq\; v_1^2 * d_3 + v_2^2 * d_3 + 2 * v_1 * v_2 * d_3$,
$v_1^2 * d_3 + v_2^2 * d_3 + 2 * v_1 * v_2 * d_3 \;\geq\; v_1^2 * d_3 + v_2^2 * d_3$,
$d_3 \geq d_1$ and $d_3 \geq d_2$,
$v_1^2 * d_3 + v_2^2 * d_3 \;\geq\; v_1^2 * d_1 + v_2^2 * d_2$,
$v_3^2 * d_3 \;\geq\; v_1^2 * d_1 + v_2^2 * d_2$.

Derivation of Constraint Class. A capability closely related to sort derivation is the derivation of constraint class. For example, simple syntactic propositions can be used to conclude that (1),(3),(4),(5), and (6) are linear constraints, (2) is a quadratic constraint, and (7) is a cubic constraint. Automated derivation of a constraint's class is useful in determining what algorithms can be applied to characterize the solution space for a set of constraints.

In its most simple incarnation, this capability can be accomplished through a trivial, superficial analysis of the syntactic structure of an expression.

Assert $d_1 = d_2 + d_3$ is a linear constraint.
Assert $d_1 * v_1 \leq d_2 * v_2 + d_3 * v_3$ is a quadratic constraint.

Evaluation-based Verification of Relationships between Constraints. It is also possible to leverage the capabilities of the automated assistant in evaluating constraints on specified ranges. The automated assistant can call appropriate constraint solvers as subroutines in order to accomplish this, or it can simply act as a calculator and perform an exhaustive evaluation of all possibilities within fixed ranges for the parameters involved.

Consider a *crossing* junction that has two incoming links (call them 1 and 2) and two outgoing links (call them 3 and 4). The corresponding gadget (call it $\mathscr{A}_{\mathbf{X}}$) again has a single node, call it **X**, defined as follows:

$$
\begin{aligned}
N &= \{\mathbf{X}\}, \mathsf{In} = \langle 1,2\rangle, Q = \varnothing, \mathsf{Out} = \langle 3,4\rangle \\
\mathsf{Par} &= \{1 \mapsto v_1 \cdot d_1, 2 \mapsto v_2 \cdot d_2, 3 \mapsto v_3 \cdot d_3, 4 \mapsto v_4 \cdot d_4\} \\
\mathsf{Con} &= \mathsf{Con}_{\text{nodes}} \cup \mathsf{Con}_{\text{links}} \qquad \text{where} \\
&\qquad \mathsf{Con}_{\text{nodes}} = \{\text{regulating traffic through } \mathbf{X}\} \\
&\qquad \mathsf{Con}_{\text{links}} \;= \{\text{lower/upper bounds on } v_1, d_1 \ldots\}
\end{aligned}
$$

Suppose all traffic entering through link 1 must exit through link 3 and at the same velocity, and all traffic entering through link 2 must exit through link 4 and at the same velocity. This is expressed by four constraints:

$$v_1 = v_3; \quad v_2 = v_4; \quad d_1 = d_3; \quad d_2 = d_4$$

If the total density of entering traffic, namely $d_1 + d_2$, exceeds a "jam density" that makes the two entering traffics block each other, there will be backups. We therefore presume there is an upper bound, say 10, on $d_1 + d_2$ below which the two traffics do not impede each other and there are no backups as a result:

$$d_1 + d_2 \leqslant 10$$

Below a total density of 10, we can imagine that the two incoming traffics are sparse enough so that they smoothly alternate taking turns to pass through the crossing junction.

We can perform a simple verification that $\{0,\ldots,5\}$ is a valid range for the components d_i simply by asserting the claim within the automated assistant.

Assert for all $d \in \{0,\ldots,5\}^4$,
        $d_1 = d_3$, $d_2 = d_4$
    implies that
        $d_1 + d_2 \leq 10$, $d_3 \in \{0,\ldots,5\}$ and $d_4 \in \{0,\ldots,5\}$.

4 Related Work and Conclusion

We reviewed the formalism underlying the NetSketch DSL, which can be used to model and assemble CFNs, and to reason in a compositional manner about constraints on flows through these networks. We have also illustrated how an automated assistant can help a domain expert using the NetSketch DSL to define modules within a domain application. The assistant can help her reason about the dimensionality of constraints, algebraic relationships between constraints, and the classification of constraint sets.

Our formalism for reasoning about constrained-flow networks was inspired by and based upon formalisms for reasoning about programs developed over the decades within the programming languages community. While our work focuses in particular on networks and constraints on flows, there is much relevant work in the community addressing the general problem of reasoning about distributed programs. However, most previously proposed systems for reasoning in general about the behavior of distributed programs (Process algebra [3], Petri nets [25], Π-calculus [22], finite-state models [18, 19], and model checking [11]) rely upon the retention of details about the *internals* of a system's components in assessing their interactions with one another. While this affords these systems great expressive power, that expressiveness necessarily carries with it a burden of complexity. There has also been a large interest in applying custom type systems to domain specific languages (which peaked in the late nineties, *e.g.*, the USENIX Conference on Domain-Specific Languages (DSL) in 1997 and 1999). Later type systems have been used to bound other resources such as expected heap space usage (*e.g.*, [10]).

Our work also shares the motivation, philosophy, and approach of earlier attempts to define DSLs for component-based design [2], [29]. Our work is distinguished by a recognition of the need for network holes, and an emphasis on the distinction between small module design and large network design.

One of the essential activities our formalism aims to support is reasoning about and finding solution ranges for sets of constraints that describe properties of a network. In its most general form, this is known as the *constraint satisfaction problem* and is widely studied [30]. However, most approaches consider a homogenous set of constraints of a particular class rather than a potentially varied collection.

One variant of the constraint satisfaction problem relevant to our work involves only linear constraints. Finding solutions respecting collections of linear constraints is a classic problem that has been considered in a large variety of work over the decades. There exists a great deal of established material [26] including many documented algorithms and many analyses of practical considerations.

The work in this paper extends and generalizes our earlier work in TRAFFIC (*Typed Representation and Analysis of Flows For Interoperability Checks* [4]), and complements our earlier work in CHAIN (*Canonical Homomorphic Abstraction of Infinite Network protocol compositions* [8]).

The accessible interface of the automated assistant utilized in this work reflects the design principles of other formal verification systems such as `Tutch` [1] and Scunak [9]. The need for natural interfaces (both superficial and functional) in automated verification has been recognized to varying degrees by the designers of the `Tutch` proof checker [1], the Scunak mathematical assistant system [9], the ForTheL language and SAD proof assistant [31], the EPGY Theorem-Proving Environment [21], the ΩMEGA proof verifier [28], and in the work of Sieg and Cittadini [27]. The ontology-oriented, lightweight verification capabilities of the automated assistant are inspired by work in the assembly of large-scale formal and semi-formal ontologies [24].

References

[1] A. Abel, B. Chang, and F. Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic, 2001.

[2] L. d. Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165, London, UK, 2001. Springer-Verlag.

[3] J. Baeten and W. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[4] A. Bestavros, A. Bradley, A. Kfoury, and I. Matta. Typed Abstraction of Complex Network Compositions. In *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP'05)*, Boston, MA, November 2005.

[5] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: The Formal Framework. Technical Report BUCS-TR-2009-029, CS Dept., Boston University, October 1 2009.

[6] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: Tool and Use Cases. In *Proceedings of CRTS'09: The IEEE/RTSS Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Washington D.C., December 2009.

[7] A. Bestavros, A. Kfoury, A. Lapets, and M. Ocean. Safe Compositional Network Sketches: The Formal Framework. In *Proceedings of HSCC'10: The 13th ACM International Conference on Hybrid Systems: Computation and Control (in conjunction with CPSWEEK)*, Stockholm, Sweden, April 2010.

[8] A. Bradley, A. Bestavros, and A. Kfoury. Systematic Verification of Safety Properties of Arbitrary Network Protocol Compositions Using CHAIN. In *Proceedings of ICNP'03: The 11th IEEE International Conference on Network Protocols*, Atlanta, GA, November 2003.

[9] C. E. Brown. Verifying and Invalidating Textbook Proofs using Scunak. In *Mathematical Knowledge Management, MKM 2006*, pages 110–123, Wokingham, England, 2006.

[10] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, pages 185–197. ACM Press, 2003.

[11] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[12] R. Horst and P. Pardalos. *Handbook of Global Optimization*. Kluwer, Dordrecht, 1995.

[13] R. M. Karp. George dantzig's impact on the theory of computation. *Discrete Optimization*, 5(2):174–185, 2008.

[14] A. Lapets. Improving the accessibility of lightweight formal verification systems. Technical Report BUCS-TR-2009-015, Computer Science Department, Boston University, April 30 2009.

[15] A. Lapets and A. Kfoury. Verification with Natural Contexts: Soundness of Safe Compositional Network Sketches. Technical Report BUCS-TR-2009-030, CS Dept., Boston University, October 1 2009.

[16] A. Lapets, A. Kfoury, and A. Bestavros. Safe Compositional Network Sketches: Reasoning with Automated Assistance. Technical Report BUCS-TR-2010-001, CS Dept., Boston University, January 2010.

[17] C. Lecoutre. *Constraint Networks: Techniques and Algorithms*. John Wiley & sons, 2009.

[18] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3)(3):219–246, Sept. 1989.

[19] N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.

[20] S. Maerivoet. *Modelling traffic on motorways: state-of-the-art, numerical data analysis, and dynamic traffic assignment*. Katholieke Universiteit Leuven, PhD thesis, 2006.

[21] D. McMath, M. Rozenfeld, and R. Sommer. A Computer Environment for Writing Ordinary Mathematical Proofs. In *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*, pages 507–516, London, UK, 2001. Springer-Verlag.

[22] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (Part I and II). *Information and Computation*, (100):1–77, 1992.

[23] A. Neumaier. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica*, 13:271–369, 2004.

[24] K. Panton, C. Matuszek, D. Lenat, D. Schneider, M. Witbrock, N. Siegel, and B. Shepard. Common Sense Reasoning – From Cyc to Intelligent Assistant. In Y. Cai and J. Abascal, editors, *Ambient Intelligence in Everyday Life*, volume 3864 of *LNAI*, pages 1–31. Springer, 2006.

[25] C. A. Petri. *Communication with Automata*. PhD thesis, Univ. Bonn, 1966.

[26] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & sons, 1998.

[27] W. Sieg and S. Cittadini. Normal Natural Deduction Proofs (in Non-classical Logics). In D. Hutter and W. Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 169–191. Springer, 2005.

[28] J. H. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Proof Development with OMEGA: sqrt(2) Is Irrational. In *LPAR*, pages 367–387, 2002.

[29] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. On relational interfaces. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 67–76, New York, NY, USA, 2009. ACM.

[30] E. Tsang. A glimpse of constraint satisfaction. *Artif. Intell. Rev.*, 13(3):215–227, 1999.

[31] K. Verchinine, A. Lyaletski, A. Paskevich, and A. Anisimov. On Correctness of Mathematical Texts from

a Logical and Practical Point of View. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 583–598, Berlin, Heidelberg, 2008. Springer-Verlag.